Efficient Representations for Set-Sharing Analysis

Eric Trias, Jorge Navas, Elena S. Ackley, Stephanie Forrest, and M.Hermenegildo

The Set-Sharing domain has been widely used to infer at compile-time interesting properties of logic programs such as occurs-check reduction, automatic parallelization, and finite-tree analysis. However, performing abstract unification in this domain requires a closure operation that increases the number of sharing groups exponentially. Much attention has been given in the literature to mitigating this key inefficiency in this otherwise very useful domain. In this paper we present a novel approach to Set-Sharing: we define a new representation that leverages the complement (or negative) sharing relationships of the original sharing set, without loss of accuracy. Intuitively, given an abstract state $sh_{\mathcal{V}}$ over the finite set of variables of interest $\mathcal{V}$, its negative representation is $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. Using this encoding during analysis dramatically reduces the number of elements that need to be represented in the abstract states and during abstract unification as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. To further compress the number of elements, we express the set-sharing relationships through a set of ternary strings that compacts the representation by eliminating redundancies among the sharing sets. Our experimental evaluation shows that our approach can compress the number of relationships, reducing significantly the memory usage and running time of all abstract operations, including abstract unification.

1 Introduction

In abstract interpretation [12] of logic programs *sharing* analysis has received considerable attention. Two or more variables in a logic program are said to *share* if in some execution of the program they are bound to terms that contain a common variable. A variable in a logic program is said to be *ground* if it is bound to a term that does not contain free variables in all possible executions of the program. *Set-Sharing* is an important type of combined sharing and groundness analysis. It was

originally introduced by Jacobs and Langen [19, 21] and its abstract values are sets of sets of variables that keep track in a compact way of the sharing patterns among variables.

Example 1.1 (Set-Sharing abstraction). Let $V = \{X_1, X_2, X_3, X_4\}$ be a set of variables. The abstraction in Set-Sharing of a substitution such as $\theta = \{X_1 \mapsto f(U_1, U_2, V_1, V_2, W_1), X_2 \mapsto g(V_1, V_2, W_1), X_3 \mapsto g(W_1, W_1), X_4 \mapsto a\}$ will be $\{\{X_1\}, \{X_1, X_2\}, \{X_1, X_2, X_3\}\}$. Sharing group $\{X_1\}$ in the abstraction represents the occurrence of run-time variables U_1 and U_2 in the concrete substitution, $\{X_1, X_2\}$ represents V_1 and V_2, and $\{X_1, X_2, X_3\}$ represents W_1. Note that X_4 does not appear in the sharing groups because X_4 is ground. Note also that the number of (occurrences of) shared run-time variables is abstracted away.

Set-Sharing has been used to infer several interesting properties and perform optimization and verification of programs at compile-time, most notably but not limited to: occurs-check reduction (e.g., [31]), automatic parallelization (e.g., [29, 28, 7]), and finite-tree analysis (e.g., [2]). The accuracy of Set-Sharing has been improved by extending it with other kinds of information, the most relevant being *freeness* and *linearity* information [27, 19, 28, 10, 17], and also information about *term structure* [28, 20, 4, 26]. Sharing in combination with other abstract domains has also been studied [9, 15, 11]. The significance of Set-Sharing is that it keeps track of sharing among sets of variables more accurately than other abstract domains such as e.g. *Pair-Sharing* [31] due to better groundness propagation and other factors that are relevant in some of its applications [6]. In addition, Set-Sharing has attracted much attention [8, 11, 3, 6] because its algebraic properties allow elegant encodings into other efficient implementations (e.g., *Reduced Ordered Binary Decision Diagrams*, *ROBDDs* [5]). In [29, 28], the first comparatively efficient algorithms were presented for performing the basic operations needed for implementing set sharing-based analyses.

However, Set-Sharing has a key computational disadvantage: the *abstract unification* (*amgu*, for short) implies potentially exponential growth in the number of sharing groups due to the *up-closure* (also called *star-union*) operation which is the heart of that operation. Considerable attention has been given in the literature to reducing the impact of the complexity of this operation. In [32], Zaffanella et al. extended the Set-Sharing domain for inferring pair-sharing from a set of sets of variables to a pair of sets of sets of variables in order to support widening. The key concept is that the set of sets in the first component (called *clique*) is reinterpreted as representing all sharing groups that are contained within it. Although significant efficiency gains are achieved, this approach loses precision with respect to the original Set-Sharing. A similar approach is followed in [30] but for

inferring set-sharing in a *top-down* framework. Other relevant work was presented in [23] in which the up-closure operation was delayed and full sharing information was recovered lazily. However, this interesting approach shares some of the disadvantages of Zaffanella's widening. Therefore, the authors refined the idea in [22] reformulating the amgu in terms of the *closure under union* operation, collapsing those closures to reduce the total number of closures and applying them to smaller descriptions without loss of accuracy. In [11] the authors show that Jacobs and Langen's sharing domain is isomorphic to the dual negative of *Pos* [1], denoted by $\overline{coPos}$. This insight improved the understanding of sharing analysis, and led to an elegant expression of the combination with groundness dependency analysis based on the reduced product of Sharing and Pos. In addition, this work pointed out the possible implementation of $\overline{coPos}$ through ROBDDs leading to more efficient implementations of Set-Sharing analyses, although this point was not investigated further therein.

In this paper we introduce a novel approach to Set-Sharing: we define a new representation that leverages the complement (or negative) sharing relationships of the original sharing set, without loss of accuracy. Intuitively, given an abstract state $sh_{\mathcal{V}}$ over the finite set of variables of interest $\mathcal{V}$, its negative representation is $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. Using this encoding during analysis dramatically reduces the number of elements that need to be represented in the abstract states and during abstract unification as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. To further compress the number of elements, we express the set-sharing relationships through a set of ternary strings that compacts the representation by eliminating redundancies among the sharing sets. It is important to notice that our work is not based on [11]. Although they define the dual negated positive Boolean functions, $\overline{coPos}$ does not represent the entire complement of the positive set. Moreover, they do not use $\overline{coPos}$ as a means of compressing relationships but as a way of representing Sharing through Boolean functions. We also represent Sharing through Boolean functions, but that is where the similarity ends.

In the remainder of the paper we first describe the Jacobs and Langen's Set-Sharing domain, bSH, adapted for handling binary strings (Section 2) and we extend it in Section 3 to tSH, a more compact representation using a ternary encoding. In Section 4, we use tSH for encoding the complement (or negative) of the original Set-Sharing, $tNSH$. Finally, we report results from an experimental evaluation of these representations in Section 5 and conclude in Section 6.

2 Set-Sharing Encoded by Binary Strings

The presentation here follows that of [32, 11], since the notation used and the abstract unification operation obtained are rather intuitive, but adapted for handling binary strings rather than sets of sets of variables. Therefore, unless otherwise stated, here and in the rest of paper we will represent the set-sharing domain using a set of strings rather than a set of sets of variables.

Example 2.1 (Binary encoding of sharing relationships). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_1, X_2\}, \{X_1, X_2, X_3\}\}$ be a sharing set. Assume the following order among variables: $X_1 \prec X_2 \prec X_3 \prec X_4$. Then, we can encode each sharing group into a binary string using the algorithm described in Figure 1. In this example, the result of mapping sh into a set of binary strings is $bsh = \{1000, 1100, 1110\}$.

```
BinaryEncoding(sh, V)
  bsh ← ∅
  foreach sg ∈ sh
    foreach i-th variable of V
      if the i-th variable of V appears in sg then
        s[i] ← 1
      else
        s[i] ← 0
    bsh ← bsh ∪ {s}
  return bsh
```

Figure 1: Simple algorithm for encoding binary sharing relationships

Definition 2.1 (Binary sharing domain, bSH). Let alphabet $\Sigma = \{0, 1\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in arbitrary order, and Σ^l the finite set of all strings over Σ with length l, $0 \leq l \leq |\mathcal{V}|$. Let $bSH^l = \wp^0(\Sigma^l)$ be the *proper power set* (i.e., $\wp(\Sigma^l) \setminus \{\emptyset\}$) that contains all possible combinations over Σ with length l. Then, the *binary sharing domain* is defined as $bSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} bSH^l$. ∎

2.1 Notation

Let $\mathcal{F}$ and $\mathcal{P}$ be sets of ranked (i.e., with a given arity) functors of interest; e.g., the function symbols and the predicate symbols of a program. We will use $Term$

to denote the set of terms constructed from $\mathcal{V}$ and $\mathcal{F} \cup \mathcal{P}$. Although somehow unorthodox, this will allow us to simply write $g \in Term$ whether g is a term or a predicate atom, since all our operations apply equally well to both classes of syntactic objects. We will denote by $\hat{t}$ the binary representation of the set of variables of $t \in Term$ according to a particular order among variables. Since $\hat{t}$ will be always used through a bitwise operation with some string of length l, the length of $\hat{t}$ must be l. If not, $\hat{t}$ is adjusted with 0's in those positions associated with variables represented in the string but not in t.

2.2 Abstract Operations

Definition 2.2 (Binary relevant sharing $rel(bsh, t)$ and irrelevant sharing $irrel(bsh, t)$). Given $t \in Term$, the set of binary strings in $bsh \in bSH^l$ of length l that are relevant with respect to t is obtained by a function $rel(bsh, t) : bSH^l \times Term \rightarrow bSH^l$ defined as:

$$rel(bsh, t) = \{s \mid s \in bsh, (s \bigwedge \hat{t}) \neq 0^l\}$$

where $\bigwedge$ represents the bitwise AND operation and 0^l is the all-zeros string of length l. Consequently, the set of binary strings in $bsh \in bSH^l$ that are *irrelevant with respect to t is a function* $irrel(bsh, t) : bSH^l \times Term \rightarrow bSH^l$ *where* $irrel(bsh, t)$ *is the complement of* $rel(bsh, t)$*, i.e.,* $bsh \setminus rel(bsh, t)$. ■

Definition 2.3 (Binary cross-union, $\bowtie$). Given $bsh_1, bsh_2 \in bSH^l$, their *cross-union* is a function $\bowtie : bSH^l \times bSH^l \rightarrow bSH^l$ defined as

$$bsh_1 \bowtie bsh_2 = \{s \mid s = s_1 \bigvee s_2, s_1 \in bsh_1, s_2 \in bsh_2\}$$

where $\bigvee$ represents the bitwise OR operation. ■

Definition 2.4 (Binary up-closure, $(.)^*$). Let l be the length of strings in $bsh \in bSH^l$, then the *up-closure* of bsh, denoted bsh^* is a function $(.)^* : bSH^l \rightarrow bSH^l$ that represents the smallest superset of bsh such that $s_1 \bigvee s_2 \in bsh^*$ whenever $s_1, s_2 \in bsh^*$:

$$bsh^* = \{s \mid \exists n \geq 1 \; \exists t_1, \ldots, t_n \in bsh, s = t_1 \bigvee \ldots \bigvee t_n\}$$

■

Definition 2.5 (Binary abstract unification, $amgu$). The abstract unification is a function $amgu : \mathcal{V} \times Term \times bSH^l \rightarrow bSH^l$ defined as

$$amgu(x, t, bsh) = irrel(bsh, x = t) \cup (rel(bsh, x) \bowtie rel(bsh, t))^*$$

■

Example 2.2 (Binary abstract unification). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}\}$ be a sharing set. Assume the following order among variables: $X_1 \prec X_2 \prec X_3 \prec X_4$. Then, we can easily encode each sharing group $sg \in sh$ into a binary string s such that $s[i] = 1$, $(1 \leq i \leq |sg|)$ if and only if the i-th variable of $\mathcal{V}$ appears in sg. In this example, sh is encoded as the following set of binary strings $bsh = \{1000, 0100,$-$0010, 0001\}$. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$
\begin{array}{lcl}
A = rel(bsh, X_1) & = & \{1000\} \\
B = rel(bsh, f(X_2, X_3)) & = & \{0100, 0010\} \\
A \bowtie B & = & \{1100, 1010\} \\
(A \bowtie B)^* & = & \{1100, 1010, 1110\} \\
C = irrel(bsh, X_1 = f(X_2, X_3)) & = & \{0001\} \\
amgu(X_1, f(X_2, X_3), bsh) = C \cup (A \bowtie B)^* & = & \{0001, 1100, 1010, 1110\}
\end{array}
$$

The design of the analysis must be completed by defining the following abstract operations that are required by an analysis engine: $init$ (initial abstract state), $equivalence$ (between two abstract substitutions), $join$ (defined as the union), and $project$.

Definition 2.6 (Binary initial state, $init_{bSH}$). The *initial state* $init_{bSH} : \mathcal{V} \rightarrow bSH$ describes an initial substitution given a set of variables. Assume that an initial substitution $sh \in SH$ is given by $init_{SH} : \mathcal{V} \rightarrow SH$, defined in [19]. Then, the binary initial state can be defined using the algorithm shown in Fig. 1 as:

$$init_{bSH}(\mathcal{V}) = \mathsf{BinaryEncoding}(init_{SH}(\mathcal{V}), \mathcal{V})$$

■

Definition 2.7 (Binary equivalence, $\equiv$). Given $bsh_1, bsh_2 \in bSH$, they are *equivalent* (i.e., $bsh_1 \equiv bsh_2$) if and only if

$$\forall s_1 \in bsh_1, \forall s_2 \in bsh_2, s_1 = s_2$$

■

Definition 2.8 (Binary join, $\sqcup$). Given $bsh_1, bsh_2 \in bSH$, the *join* function $\sqcup : bSH \times bSH \rightarrow \wp(bSH)$ is defined as their union:

$$bsh_1 \sqcup bsh_2 = bsh_1 \cup bsh_2$$

■

Definition 2.9 (Binary projection, $bsh|_t$). The *binary projection* is a function $bsh|_t$: $bSH^l \times Term \rightarrow bSH^k$ ($k \leq l$) that removes the i-th positions from all strings (of length l) in $bsh \in bSH^l$, if and only if the i-th positions of $\hat{t}$ (denoted by $\hat{t}[i]$) is 0, and it is defined as

$$bsh|_t = \{s' \mid s \in bsh, s' = \pi(s,t)\}$$

where $\pi(s,t)$ is the binary string projection defined as

$$\pi(s,t) = \begin{cases} \epsilon, & \text{if } s = \epsilon, \text{ the empty string} \\ \pi(s',t), & \text{if } s = s'a_i \text{ and } \hat{t}[i] = 0 \\ \pi(s',t)a_i, & \text{if } s = s'a_i \text{ and } \hat{t}[i] = 1 \end{cases}$$

and $s'a_i$ is the concatenation of character a to string s' at position i.

■

3 Ternary Set-Sharing

In this section, we introduce a more efficient representation for the Set-Sharing domain defined in Sec. 2 to accommodate a larger number of variables for analysis. We extend the binary string encoding discussed above to the ternary alphabet $\Sigma_* = \{0, 1, *\}$, where the $*$ symbol denotes both 0 and 1 bit values. This representation effectively compresses the number of elements in the set into fewer strings without changing what is represented (i.e., without loss of accuracy). To handle the ternary alphabet, we redefine the binary operations covered in Sec. 2.

Definition 3.1 (Ternary Sharing Domain, tSH). Let alphabet $\Sigma_* = \{0, 1, *\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in an arbitrary order as in Def. 2.1, and Σ_*^l the finite set of all strings over Σ_* with length l, $0 \leq l \leq |\mathcal{V}|$. Then, $tSH^l = \wp^0(\Sigma_*^l)$ and hence, the *ternary sharing domain* is defined as $tSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} tSH^l$. ■

Prior to defining how to transform the binary string representation into the corresponding ternary string representation, we introduce two core definitions, Def. 3.2 and Def. 3.3, for comparing ternary strings. These operations are essential for the conversion and set operations. In addition, they are used to eliminate redundant strings within a set and to check for equivalence of two ternary sets containing different strings.

Definition 3.2 (Match, $\mathcal{M}$). Given two ternary strings, $x, y \in \Sigma_*^l$, of length l, *match* is a function $\mathcal{M} : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$, such that $\forall i\ 1 \leq i \leq l$,

$$x\mathcal{M}y = \begin{cases} \mathsf{true}, \text{if } (x[i] = y[i]) \vee (x[i] = *) \vee (y[i] = *) \\ \mathsf{false}, \text{otherwise} \end{cases}$$

■

Definition 3.3 (Subsumed_By $\trianglelefteq$ and Subsumed_In $\trianglelefteqq$). Given two ternary strings $s_1, s_2 \in \Sigma_*^l$, $\trianglelefteq : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$ is a function such that $s_1 \trianglelefteq s_2$ if and only if every string matched by s_1 is also matched by s_2. More formally, $s_1 \trianglelefteq s_2 \iff \forall s \in tSH^l,\ if\ s_1 \mathcal{M} s\ \ then\ \ s_2 \mathcal{M} s$. For convenience, we augment this definition to deal with sets of strings. Given a ternary string $s \in \Sigma_*^l$ and a ternary sharing set, $tsh \in tSH^l$, $\trianglelefteqq : \Sigma_*^l \times tSH^l \rightarrow \mathcal{B}$ is a function such that $s \trianglelefteqq tsh$ if and only if there exists some element $s' \in tsh$ such that $s \trianglelefteq s'$. ■

Figure 2 gives the pseudo code for an algorithm which converts a set of binary strings into a set of ternary strings. The function Convert evaluates each string of the input and attempts to introduce $*$ symbols using PatternGenerate, while eliminating redundant strings using ManagedGrowth.

PatternGenerate evaluates the input string bit-by-bit to determine where the $*$ symbol can be introduced. The number of $*$ symbols introduced depends on the sharing set represented and k, the desired minimum number of specified bits, where $0 \leq k \leq l$ (the string length). For a given set of strings of length l, parameter k controls the compression of the set. For $k = l$ (all bits specified), there is no compression and $tsh = bsh$. For a non-empty bsh, $k = 1$ introduces the maximum number of $*$ symbols. For now, we will assume that $k = 1$, and some experimental results in Section 5 will show the best overall k value for a given l. The Specified function returns the number of specified bits (0 or 1) in x.

ManagedGrowth checks if the input string y subsumes other strings from tsh. If no redundant string exists, then y is appended to tsh only if y itself is not redundant to an existing string in tsh. Otherwise, y replaces all the redundant strings.

Example 3.1 (Conversion from bSH to tSH). Let $\mathcal{V}$ be the set of variables of interest with the same order as Example 2.2. Assume the following sharing set of binary strings $bsh = \{1000, 1001, 0100, 0101, 0010, 0001\}$. Then, a ternary string representation produced by applying Convert is $tsh = \{100*, 0010, 010*, *001\}$. There can be a certain level of redundancy in the representation, a subject that will be discussed further in Section 5.

The example above begins with Convert(bsh,$k = 1$).

1. Since $tsh = \emptyset$ initially (line 1), the first string 1000 is appended to tsh, so $tsh = \{1000\}$.

2. Next, 1001 from bsh is evaluated. In PatternGenerate, with x' at iteration i (denoted as x'_i), $i = 3$ and $b_3 = 1$, we test $x'_3 = 1000$ if the i^{th} position of

```
0  Convert(bsh, k)
1  tsh ← ∅
2  foreach s ∈ bsh
3      y ← PatternGenerate(tsh, s, k)
4      tsh ← ManagedGrowth(tsh, y)
5  return tsh
```

```
10 PatternGenerate(tsh, x, k)
11 m ← Specified(x)
12 i ← 0
13 x' ← x
14 l ← length(x)
15 while m > k and i < l
16     Let b_i be the value of x' at position i
17     if b_i = 0 or b_i = 1 then
18         x' ← x' with position i replaced by b̄_i
19         if x' ⋦ tsh then
20             x' ← x' with position i replaced by *
21         else
22             x' ← x' with position i replaced by b_i
23     m ← Specified(x')
24     i ← i + 1
25 return x'
```

```
30 ManagedGrowth(tsh, y)
31 S_y = {s | s ∈ tsh, s ⋦ y}
32 if S_y = ∅ then
33     if y ⋦̸ tsh then
34         append y to tsh
35 else
36     remove S_y from tsh
37     append y to tsh
38 return tsh
```

Figure 2: A deterministic algorithm for converting a set of binary strings bsh into a set of ternary strings tsh, where k is the desired minimum number of specified bits (non-$*$) to remain.

x can be replaced with a $*$ (line 15-24). In this case, since $x'_3 \trianglelefteq tsh$ (line 19), $x'_3 = 100*$ is returned (line 25). Next, ManagedGrowth evaluates 100* and since it subsumes 1000 ($S_y = \{1000\}$), 100* replaces 1000 leaving $tsh = \{100*\}$ (line 38).

3. The process continues with PatternGenerate($\{100*\}$,0100) (line 3). In PatternGenerate, since $x'_0 \not\trianglelefteq tsh$, $x'_1 \not\trianglelefteq tsh$, $x'_2 \not\trianglelefteq tsh$, and $x'_3 \not\trianglelefteq tsh$, we reset each i^{th} bit to its original value (line 22) and $x' = x = 0100$ is returned. Next, ManagedGrowth($\{100*\}$,0100) is called and since 0100 is not redundant to any string in tsh, it is appended to tsh resulting in $tsh = \{100*,0100\}$.

4. The process continues with PatternGenerate($\{100*,0100\}$,0101). In PatternGenerate, when $x'_3 = 0100$ and since $x'_3 \trianglelefteq tsh$, then $x'_3 = 010*$ is

returned. ManagedGrowth({100*, 0100}, 010*) is called next and since 010* subsumes 0100 in tsh, it is replaced leaving tsh ={100*,010*} (line 38).

5. The process continues similarly, for the remaining input strings in bsh obtaining the final result of $tsh = \{100*, 0010, 010*, *001\}$.

Next, we redefine the binary string operations to account for the $*$ symbol in a ternary string. Note that since the ternary representation extends the binary alphabet (i.e., binary is a subset of the ternary alphabet), ternary operations can also operate over strictly binary strings. For simplicity, we will overload certain operators to denote operations involving both binary and ternary strings.

Definition 3.4 (Ternary-or $\bigvee$ and Ternary-and $\bigwedge$). Given two ternary strings, $x, y \in \Sigma_*^l$ of length l, *ternary-or* and *ternary-and* are two bitwise-or functions defined as $\bigvee, \bigwedge : \Sigma_*^l \times \Sigma_*^l \rightarrow \Sigma_*^l$ such that $z = x \bigvee y$ and $w = x \bigwedge y, \forall i\ 1 \leq i \leq l$, where:

$$z[i] = \begin{cases} * & \text{if } (x[i] = * \wedge y[i] = *) \\ 0 & \text{if } (x[i] = 0 \wedge y[i] = 0) \\ 1 & otherwise \end{cases} \qquad w[i] = \begin{cases} * & \text{if } (x[i] = * \wedge y[i] = *) \\ 1 & \text{if } (x[i] = 1 \wedge y[i] = 1) \\ & \vee\ (x[i] = 1 \wedge y[i] = *) \\ & \vee\ (x[i] = * \wedge y[i] = 1) \\ 0 & otherwise \end{cases}$$

■

Definition 3.5 (Ternary set intersection, $\cap$). Given $tsh_1, tsh_2 \in tSH^l$, $\cap : tSH^l \times tSH^l \rightarrow tSH^l$ is defined as

$$tsh_1 \cap tsh_2 = \{r \mid r = s1 \bigwedge s2, s1 \mathcal{M} s2, s1 \in tsh1, s2 \in tsh2\}$$

■

For convenience, we define two binary patterns, 0-mask and 1-mask, in order to simplify further operations. The former takes an l-length binary string s and returns a set with a single string having a 0 where $s[i] = 1$ and $*$'s elsewhere, $\forall i\ 1 \leq i \leq l$. The latter takes also an l-length binary string s, but returns a set of strings with a 1 where $s[i] = 1$ and $*$'s elsewhere, $\forall i\ 1 \leq i \leq l$. For instance, 0-mask(0110) and 1-mask(0110) return $\{*00*\}$ and $\{*1**, **1*\}$, respectively.

Definition 3.6 (Ternary relevant sharing $rel(tsh, t)$ and irrelevant sharing $irrel(tsh, t)$). Given $t \in Term$ with length l and $tsh \in tSH^l$ with strings of length l, the set of strings in tsh that are *relevant* with respect to t is obtained by a function $rel(tsh, t) : tSH^l \times Term \rightarrow tSH^l$ defined as

$$rel(tsh, t) = tsh \cap \textsf{1-mask}(\hat{t})$$

In addition, $irrel(tsh, t)$ is defined as

$$irrel(tsh, t) = (tsh \cap \textsf{1-mask}(\bar{\hat{t}})) \cap \textsf{0-mask}(\hat{t})$$

■

Ternary cross-union, $\bowtie$, and ternary up-closure, $(.)^*$, operations are as defined in Def. 2.3 and in Def. 2.4, respectively, except the binary version of the bitwise OR operator is replaced with its ternary counterpart defined in Def. 3.4 in order to account for the $*$ symbol. In addition, the ternary abstract unification ($amgu$) is defined exactly as the binary version, Def.2.5, using the corresponding ternary definitions.

Example 3.2 (Ternary abstract unification). Let $tsh = \{100*, 010*, 0010, *001\}$ as in Example 3.1. Consider again the analysis of $X_1 = f(X_2, X_3)$, the result is:

$$\begin{array}{lcl}
A = rel(tsh, X_1) & = & \{100*\} \\
B = rel(tsh, f(X_2, X_3)) & = & \{010*, 0010\} \\
A \bowtie B & = & \{110*, 101*\} \\
(A \bowtie B)^* & = & \{110*, 101*, 111*\} \\
C = irrel(tsh, X_1 = f(X_2, X_3)) & = & \{0001\} \\
amgu(X_1, f(X_2, X_3), tsh) = C \cup (A \bowtie B)^* & = & \{0001, 110*, 101*, 111*\}
\end{array}$$

Definition 3.7 (Ternary initial state, $init_{tSH}$). The *initial state* $init_{tSH} : \mathcal{V} \times \mathcal{I}^+ \rightarrow tSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming the binary initial state operation defined as $init_{bSH} : \mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$, the ternary initial state can be defined using the Convert algorithm in Fig. 2 as:

$$init(\mathcal{V}, k) = \textsf{Convert}(init_{bSH}(\mathcal{V}), k)$$

■

Definition 3.8 (Ternary equivalence, $\equiv$). Given $tsh_1, tsh_2 \in tSH^l$, the sets are *equivalent* if and only

$$\forall t_1 \in tsh_1, \forall s_1 \trianglelefteq t_1, s_1 \trianglelefteqslant tsh_2) \ \wedge \ (\forall t_2 \in tsh_2, \forall s_2 \trianglelefteq t_2, s_2 \trianglelefteqslant tsh_1)$$

■

The ternary join is defined as its binary counterpart, i.e., union. Finally, the ternary projection, $tsh|_t$, is defined similarly as binary projection, see Def. 2.9. However, the projection domain and range is extended to accommodate the $*$ symbol. So, the

function definition remains the same except that *ternary* string projection is now defined as a function $\pi(s,t)$: $\Sigma_*^l \times Term \rightarrow \Sigma_*^k$, $k \leq l$. For example, let $tsh = \{100*, 010*, 0010, *001\}$ as in Example 3.1. Then, the projection of tsh over the term $t = f(X_1, X_2, X_3)$ is $tsh|_t = \{100, 010, 001\}$. Note that since all zeros is meaningless in a set-sharing representation, it is not included here.

4 Negative Ternary Set-Sharing

In this section, we describe a further step using the ternary representation discussed in the previous section.* In certain cases, a more compact representation of sharing relationships among variables can be captured equivalently by working with the complement (or negative) set of the original sharing set. A ternary string t can either be *in* or *not in* the set $tsh \in tSH$. This mutual exclusivity together with the finiteness of $\mathcal{V}$ allows for checking t's membership in tsh by asking if t is in tsh, or, equivalently, if t is *not* in its complement, $\overline{tsh}$. The same reasoning is applicable to binary strings (i.e., bSH). Given a set of l-bit binary strings, its complement or negative set contains *all* the l-bit ternary strings *not* in the original set. Therefore, if the cardinality of a set is greater than half of the maximum size (i.e., $2^{|\mathcal{V}|-1}$), then the size of its complement will not be greater than $2^{|\mathcal{V}|-1}$. It is this size differential that we exploit. In Set-Sharing analysis, as we consider programs with larger numbers of variables of interest, the potential number of sharing groups grows exponentially, toward $2^{|\mathcal{V}|}$, whereas the number of sharing groups not in the sharing set decreases toward 0.

The idea of a negative set representation and its associated algorithms extends the work by Esponda et al. in [13, 14]. In that work, a negative set is generated from the original set in a similar manner to the conversion algorithms shown in Figs. 2 and 3. However, they produce a negative set with unspecified bits in random positions and with less emphasis on managing the growth of the resulting set. The technique was originally introduced as a means of generating Boolean satisfiability (SAT) formulas where, by leveraging the difficulty of finding solutions to hard SAT instances, the contents of the original set are obscured without using encryption [13]. In addition, these hard-to-reverse negative sets are still able to answer membership queries efficiently while remaining intractable to reverse (i.e., to obtain the contents of the original set). In this paper, we are not interested in this security property, however, and use the negative approach simply to address the efficiency issues faced by the traditional Set-Sharing domain.

*Note that we could have also used the binary representation described in Sec. 2 but we chose the ternary encoding in order to achieve more compactness.

The conversion to the negative set can be accomplished using the two algorithms shown in Figure 3. NegConvert uses the Delete operation to remove input strings of the set sh from $\mathcal{U}$, the set of all l-bit strings $\mathcal{U} = \{*^l\}$, and then, the Insert operation to return $\mathcal{U} \setminus sh$ which represents all strings *not* in the original input. Alternatively, NegConvertMissing uses the Insert operation directly to append each string *missing* from the input set to an empty set resulting in a representation of all strings *not* in the original input. Although as shown in Table 1 both algorithms have similar complexities, depending on the size of the original input it may be more efficient to find all the strings missing from the input and transform them with NegConvertMissing, rather than applying NegConvert to the input directly. Note that the resulting negative set will use the same ternary alphabet described in Def. 3.1. For clarity, we will denote it by $tNSH$ such that $tNSH \equiv tSH$.

For simplicity, we describe only NegConvert since NegConvertMissing uses the same machinery. Assume a transformation from bsh to $tnsh$ calling NegConvert with $k = 1$. We begin with $tnsh = \mathcal{U} = \{****\}$ (line 1), then incrementally Delete each element of bsh from $tnsh$ (line 2-3). Delete removes all strings matched by x from $tnsh$ (line 11-12). If the set of matched strings, D_x, contains unspecified bit values (* symbols), then all string combinations *not* matching x must be re-inserted back into $tnsh$ (line 13-17). Each string y' not matching x is found by setting the unspecified bit to the opposite bit value found in $x[i]$ (line 16). Then, Insert ensures string y' has at least k specified bits (line 22-26). This is done by specifying $k - m$ unspecified bits (line 23) and appending each to the result using ManagedGrowth (line 24-26). If string x already has at least k specified bits, then the algorithm attempts to introduce more $*$ symbols using PatternGenerate (line 28) and appends it while removing any redundancy in the resulting set using ManagedGrowth (line 29).

Example 4.1 (Conversion from bSH to tNSH). Consider the same sharing set as in Example 3.1: $bsh = \{1000, 1001, 0100, 0010, 0101, 0001\}$. A negative ternary string representation is generated by applying the NegConvert algorithm to obtain {0000, 11**, 1*1*, *11*, **11}. Since a string of all 0's is meaningless in a set-sharing representation, it is removed from the set. Thus, $tnsh$ = {11**, 1*1*, *11*, **11}.

1. The first string 1000 is deleted from $\mathcal{U} = \{****\}$. So, $D_x = \{****\}$ (line 11) and $tnsh' = \emptyset$ (line 12). For each i^{th} bit of x, a new y'_i that x is evaluated for insertion into the result set. So, Insert ($\emptyset$, $y'_0 = 0***$, $k = 1$) is called (line 17). Since Specified(y') $\geq k$ and $tnsh' = \emptyset$, the result returned is $tnsh'$ ={0***} (line 27-30). For all other unspecified positions (line 14) of y, a new string is created with a bit value opposite to x_i's value, ($\overline{b_i}$). So,

```
0  NegConvert(sh, k)
1  tnsh ← U
2  foreach t ∈ sh
3      tnsh ← Delete(tnsh, t, k)
4  return tnsh
```

```
0  NegConvertMissing(bsh, k)
1  tnsh ← ∅
2  bnsh ← U \ bsh
3  foreach t ∈ bnsh
4      tnsh ← Insert(tnsh, t, k)
5  return tnsh
```

```
10 Delete(tnsh, x, k)
11 D_x ← ∀t ∈ tnsh, x ℳ t
12 tnsh' ← tnsh with D_x removed
13 foreach y ∈ D_x
14     foreach unspecified bit position q_i of y
15         if b_i (the i^th bit of x) is specified, then
16             y' ← y with position q_i replaced by b̄_i
17             tnsh' ← Insert(tnsh', y', k)
18 return tnsh'
```

```
20 Insert(tnsh, x, k)
21 m ← Specified(x)
22 if m < k then
23     P ← select (k − m) unspecified bit positions in x
24     V_P ← every possible bit assignment of length |P|
25     foreach v ∈ V_P
26         y ← x with positions P replaced by v
27         tnsh' ← ManagedGrowth(tnsh, y)
28 else
29     y ← PatternGenerate(tnsh, x, k)
30     tnsh' ← ManagedGrowth(tnsh, y)
31 return tnsh'
```

Figure 3: NegConvert, NegConvertMissing, Delete and Insert algorithms used to transform positive to negative representation; k is the desired number of specified bits (non-*'s) to remain.

Insert ({0***}, y'_1 = *1**, $k = 1$) is called next and y'_1 is appended to $tnsh'$. The process continues with y'_2 and y'_3 resulting in $tnsh$ = {0***, *1**, **1*, ***1}.

2. Next, 1001 from bsh is deleted (line 2) resulting in D_x ={***1} and $tnsh'$ = {0***, *1**, **1*} (line 11,12). Then, Insert ({0***, *1**, **1*}, y' = 0**1, $k = 1$) is called. Since 0**1 $\trianglelefteq tnsh'$, then $tnsh'$ remains unchanged.

The process continues with y'_1 =*1*1, y'_2 =**11 being subsumed by $tnsh'$; so the result returned is $tnsh = \{0***, *1**, **1*\}$.

3. Next, 0100 is deleted resulting in $tnsh = \{00**, 0**1, 11**, *1*1, **1*\}$.

4. Next, 0010 is deleted resulting in $tnsh = \{000*, 0**1, 11**, 1*1*, *11*, *1*1, **11\}$.

5. Next, 0101 is deleted resulting in $tnsh = \{000*, 00*1, 11**, 1*1*, *11*, **11\}$.

6. Finally, 0001 is deleted resulting in $tnsh = \{0000, 11**, 1*1*, *11*, **11\}$.

7. Removing the string with all 0s, we get the final $tnsh = \{11**, 1*1*, *11*, **11\}$.[†]

An alternative conversion algorithm uses NegConvertMissing. First the missing strings must be calculated from the given set. For Example 4.1, the missing strings are $\{0011, 0110, 0111, 1010, 1011, 1100, 1101, 1110, 1111\}$. NegConvertMissing begins with the first string 0011 and $tnsh = \emptyset$ resulting in $tnsh = \{0011\}$.

1. Then, Insert ($\{0011\}$, $y' = 0110$, $k = 1$) resulting in $tnsh = \{0011, 0110\}$.

2. Next, Insert ($\{0011, 0110\}$, $y' = 0111$, $k = 1$) resulting in $tnsh = \{011*, 0*11\}$.

3. Next, Insert ($\{011*, 0*11\}$, $y' = 1010$, $k = 1$) resulting in $tnsh = \{011*, 0*11, 1010\}$.

4. Next, Insert ($\{011*, 0*11, 1010\}$, $y' = 1011$, $k = 1$) resulting in $tnsh = \{011*, 0*11, 101*, *011\}$.

5. Next, Insert ($\{011*, 0*11, 101*, *011\}$, $y' = 1100$, $k = 1$) resulting in $tnsh = \{011*, 0*11, 101*, 1100, *011\}$.

6. Next, Insert ($\{011*, 0*11, 101*, 1100, *011\}$, $y' = 1101$, $k = 1$) resulting in $tnsh = \{011*, 0*11, 101*, 110*, *011\}$.

7. Next, Insert ($\{011*, 0*11, 101*, 110*, *011\}$, $y' = 1110$, $k = 1$) resulting in $tnsh = \{011*, 0*11, 101*, 110*, *011, *110\}$.

[†]Notice that $tnsh = \mathcal{U} \setminus (bsh \cup \{0000\})$.

Transformation	Time Complexity	Size Complexity
$bSH \rightarrow tSH$	$O(\|bsh\|\alpha l)$	$O(\|bsh\|)$
$bSH/tSH \rightarrow tNSH$	$O(\|bsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tnsh\|(l-m)2^{\delta})$
$tNSH \rightarrow tSH$	$O(\|tnsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tsh\|(l-m)2^{\delta})$
$bSH \rightarrow tNSH$	$O(\beta+\|bnsh\|(\alpha 2^{\delta}+1))$	$O(\|bnsh\|2^{\delta})$

Table 1: Summary of conversions: l-length strings; $\alpha = |Result| \cdot l$; if $m < k$ then $\delta = k - m$ else $\delta = 0$, where $m =$ minimum specified bits in entire set, $k =$ number of specified bits desired; $bnsh = \mathcal{U} \setminus bsh$; $\beta = O(2^l)$ time to find $bnsh$.

8. Finally, Insert ($\{011*, 0*11, 101*, 110*, *011, *110\}$, $y' = 1111$, $k = 1$) resulting in $tnsh = \{11**, 1*1*, *11*, **11\}$.

Notice that NegConvertMissing returns the same result for Example 4.1, and, in general, an equivalent negative representation.

Table 1 illustrates the different transformation functions and their complexities for a given input. Transformation $bSH \rightarrow tSH$ can be performed by the Convert algorithm described in Fig. 2. Transformations $bSH/tSH \rightarrow tNSH$ and $bSH \rightarrow tNSH$ are done by NegConvert and NegConvertMissing, respectively. Both transformations show that we can convert a positive representation into negative with corresponding difference in time and memory complexity. Depending on the size of the original input we may prefer one transformation over another. If the input size is relatively small, less than 50% of the maximum size, then NegConvert is often more efficient than NegConvertMissing. Otherwise, we may prefer to insert those strings missing in the input set. In our implementation, we continuously track the size of the relationships to choose the most efficient transformation. Finally, transformation $tNSH \rightarrow tSH$ is performed by NegConvert allowing comming back to the positive from a negative representation.

Consider now the same set of variables and order among them as in Example 4.1 but with a slightly different set of sharing groups encoded as $bsh = \{1000, 1100, 1110\}$ or $tsh = \{1*00, 1110\}$. Then, a negative ternary string representation produced by NegConvert is $tnsh = \{00**, 01**, 0*1*, 0**1, 1**1, *01*\}$. This example shows that the number of elements, or size, of the negative result, $|tnsh| = 6 > |bsh| = 3$ and $|tsh| = 2$. However, in Example 4.1 when $|bsh| = 6$, $|tnsh| = 4 < |bsh|$. This is because when $|bsh|$ is less than $2^{|\mathcal{V}|-1}$, i.e., $|bsh| = 3 < 2^3$, then its complement set must represent $(2^{|\mathcal{V}|} - |bsh|) = 13$ elements. Depending on the strings in the positive set, the size of the negative result may indeed be greater. This is a good illustration of how selecting the appropriate set-sharing representation will affect the size of the converted result. Thus, the size

of the original sharing set at specific program points will be used by the analysis to produce the most compact working set. The negative sharing set representation allows us to represent more variables of interest enabling larger problem instances to be evaluated.

We now define certain operations on the negative representation in order to perform abstract unification and the other abstract operations required by our engine to use the negative representation.

Definition 4.1 (Negative intersection, $\overline{\cap}$). Given two negative sets with same length strings, $tnsh_1$ and $tnsh_2$, the *Negative Intersection* returns a negative set representing the set intersection of $tnsh_1 \,\overline{\cap}\, tnsh_2$, and is defined in [14] as:

$$tnsh_1 \,\overline{\cap}\, tnsh_2 = \{x | x \in tnsh_1\} \cup \{y | y \in tnsh_2\}.$$

■

Definition 4.2 (Negative relevant sharing $\overline{rel}(tnsh, t)$ and irrelevant sharing $\overline{irrel}(tnsh, t)$) Given $t \in Term$ and $tnsh \in tNSH^l$ with strings of length l, the set of strings in $tnsh$ that are *negative relevant* with respect to t is obtained by a function $\overline{rel}(tnsh, t) : tNSH^l \times Term \rightarrow tNSH^l$ defined as:

$$\overline{rel}(tnsh, t) = tnsh \,\overline{\cap}\, \texttt{0-mask}(\hat{t}),$$

n addition, $\overline{irrel}(tnsh, t)$ is defined as:

$$\overline{irrel}(tnsh, t) = tnsh \,\overline{\cap}\, \texttt{1-mask}(\hat{t}).$$

■

Because the negative representation is the complement, it is not only more compact for large positive set-sharing instances, but also, and perhaps more importantly, it enables us to use inverse operations that are more memory- and computationally efficient than in the positive representation. However, the negative representation does have its limitations. Certain operations that are straightforward in the positive representation are $\mathcal{NP}$-Hard in the negative representation [13, 14]. A key observation given in [13] is that there is a mapping from Boolean formulae to the negative set-sharing domain such that finding which strings are not represented is equivalent to finding satisfying assignments to the corresponding Boolean formula. This is known to be an $\mathcal{NP}$-Hard problem. As mentioned before, this fact is exploited in [13] for privacy enhancing applications. The mapping is defined as follows.

Let $tnsh = \{$11**, 1*1*, *11*, **11$\}$ be the same sharing set as in Example 4.1. Its equivalent Boolean formula $\phi \equiv$ not [(x_1 and x_2) or (x_1 and x_3) or

(x_2 and x_3) or (x_3 and x_4)] is defined over the set of variables $\{x_1, x_2, x_3, x_4\}$. The formula ϕ is mapped into a negative set-sharing instance where each clause corresponds to a string and each variable in the clause is represented as a 0 if it appears negated, as a 1 if it appears un-negated, and as a * if it does not appear in the clause. By applying DeMorgan's law, we can convert ϕ to an equivalent formula in conjunctive normal form. Then, it is easy to see that a satisfying assignment of the formula such as $\{x_1 = true, x_2 = false, x_3 = false, x_4 = true\}$ corresponding to the string 1001 is not represented in the negative set-sharing instance.

Theorem 1 *A polynomial time algorithm for computing negative cross-union, $\overline{\boxtimes}$, implies $\mathcal{P}$=$\mathcal{NP}$.*

To show that negative cross-union, $\overline{\boxtimes}$, is $\mathcal{NP}$-Complete we first restate the definition of Non-Empty Self Recognition ($NESR$) shown to be $\mathcal{NP}$-Complete in [13]. Then, we use $NESR$ to show that there is no polynomial time algorithm for computing negative cross-union unless $\mathcal{P} = \mathcal{NP}$.

(Non-Empty Self Recognition, $NESR$).
INPUT: A negative set $tnsh$ of length l strings over the alphabet $\{0, 1, *\}$.
QUESTION: Does $tnsh$ represent an empty positive set bsh? In other words, does there exists a string in $\{0, 1\}^l$ not matched in $tnsh$?

The following is a proof for Theorem 1:

Proof 1 *Given a negative set $tnsh$ of length l, assume a polynomial time algorithm $\mathcal{M}$ that takes as input negative sets $tnsh_1$ and $tnsh_2$ and outputs $tnsh' = tnsh_1 \overline{\boxtimes} tnsh_2$, where $tnsh'$ represents the result of the positive cross-union of the two positive sets represented by $tnsh_1$ and $tnsh_2$.*

We construct a polynomial time algorithm for $NESR$: given any instance of $NESR$ with input $tnsh$. First, generate a positive set sh with two strings s_1 and s_2 of length l each having alternating 1's and 0's, e.g., if $l = 4$, then $sh = \{0101, 1010\}$. Convert sh to its negative set representation, nsh, using a polynomial time algorithm, i.e., letting $k = log_2(l)$ or the Prefix algorithm, see [13]. Verify that s_1 and s_2 appear in $tnsh$: if either one is missing from $tnsh$, then answer "No" ($tnsh$ is not empty, at a minimum it represents the missing string). Otherwise, both s_1 and s_2 appear in $tnsh$, but there may be some other string(s) missing from $tnsh$ ($tnsh$ is not empty). Let $\mathcal{M}$ compute $tnsh' = tnsh \overline{\boxtimes} nsh$. Now, check if both s_1 and s_2 appear in $tnsh'$: if both are missing from $tnsh'$, then answer "Yes" ($tnsh$ is empty); otherwise, answer "No".

Note that if $tnsh$ represented an empty positive set, then its negative cross-union *with another set nsh will yield a representation of the same set nsh. In*

other words, if $tnsh$ is empty and since $s1$ and $s2$ were missing from nsh, then $s1$ and $s2$ will also be missing from the result $tnsh'$. On the other hand, if $tnsh$ is not *empty (represents some string(s), other than $s1$ and $s2$, in the positive), then negative cross-union (ternary OR operation) with one of the two strings will produce a different string to s_1 or s_2 resulting in either s_1 or s_2 appearing in $tnsh'$. Thus, $\mathcal{M}$ can be used to solve $NESR$ efficiently. Since $NESR$ is $\mathcal{NP}$-Complete, then $\mathcal{P}$=$\mathcal{NP}$.*

Due to the interdependent nature of the relationship between the elements of a negative set, it is unclear how a precise negative cross-union can be accomplished without going through a positive representation. Therefore, we accomplish the negative cross-union by first identifying the represented positive strings and then applying cross-union accordingly.

Rather than iterating through all possible strings in $\mathcal{U}$ and performing cross-union on strings not in $tnsh$, we achieve a more efficient negative cross-union, $\overline{\boxtimes}$, by converting $tnsh$ to tsh first, i.e., using NegConvert from Table 1 and performing ternary cross-union on strings $t \in tsh$. In this way, the ternary representation continues to provide a compressed representation of the sharing set. Note that the negative up-closure operation, $\overline{*}$, suffers the same drawback as cross-union. Therefore, we handle it in the same way as the negative cross-union.

Definition 4.3 (Negative union, $\overline{\cup}$). Given two negative sets with same length strings, $tnsh_1$ and $tnsh_2$, the *Negative Union* returns a negative set representing the set union of $tnsh_1 \,\overline{\cup}\, tnsh_2$, and is defined in [14] as:

$$tnsh_1 \,\overline{\cup}\, tnsh_2 = \{z | (x \mathcal{M} y) \Rightarrow z = x \bigwedge y, x \in tnsh_1, y \in tnsh_2\},$$

where $\bigwedge$ is the ternary AND operator.

■

Definition 4.4 (Negative abstract unification, $\overline{amgu}$). The *negative abstract unification* is a function $\overline{amgu} : \mathcal{V} \times Term \times tNSH^l \rightarrow tNSH^l$ defined as

$$\overline{amgu}(x, t, tnsh) = \overline{irrel}(tnsh, x = t) \,\overline{\cup}\, (\overline{rel}(tnsh, x) \,\overline{\boxtimes}\, \overline{rel}(tnsh, t))^{\overline{*}},$$

■

Example 4.2 (Negative abstract unification). Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 4.1. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$
\begin{array}{lcl}
A = \overline{rel}(tnsh, X_1) & = & \{11**, 1*1*, *11*, **11, 0***\} \\
B = \overline{rel}(tnsh, f(X_2, X_3)) & = & \{11**, 1*1*, *11*, **11, *00*\} \\
A \overline{\boxtimes} B & = & \{00**, 01**, 0*0*, *00*\} \\
(A \overline{\boxtimes} B)^{\overline{*}} & = & \{01**, 0*1*, 100*\} \\
C = \overline{irrel}(tnsh, X_1 = f(X_2, X_3)) & = & \{11**, 1*1*, *11*, **11, 1***, \\
 & & *1**, **1*\} \\
 & = & \{1***, *1**, **1*\} \\
\overline{amgu}(X_1, f(X_2, X_3), tnsh) = C \,\overline{\cup}\, (A \overline{\boxtimes} B)^{\overline{*}} & = & \{01**, 0*1*, 0**0, 100*\}
\end{array}
$$

Definition 4.5 (Negative initial state, $\overline{init}$). The *negative initial state* $\overline{init} : \mathcal{V} \times \mathcal{I}^+ \rightarrow tNSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming as in Def. 3.7 the binary initial state operation $init_{bSH} : \mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$, the negative initial state can be defined using either NegConvert or NegConvert-Missing described in Fig. 3 and denoted both by $\overline{\text{Convert}}$ as follows:

$$\overline{init}(\mathcal{V}, k) = \overline{\text{Convert}}(init_{bSH}(\mathcal{V}), k)$$

■

Definition 4.6 (Negative set equivalence, $\overline{\equiv}$). Given $tnsh_1, tnsh_2 \in tNSH^l$, they are *equivalent* if and only if

$$\forall t_1 \in tnsh_1, \forall s_1 \trianglelefteq t_1, s_1 \overline{\trianglelefteq} tnsh_2) \ \wedge \ (\forall t_2 \in tnsh_2, \forall s_2 \trianglelefteq t_2, s_2 \overline{\trianglelefteq} tnsh_1)$$

■

Definition 4.7 (Negative join, $\overline{\sqcup}$). Given $tnsh_1, tnsh_2 \in tNSH^l$, the *negative join* function $\overline{\sqcup} : tNSH^l \times tNSH^l \rightarrow \wp^0(tNSH^l)$ is defined as the negative set union of the two sets, i.e.,

$$tnsh_1 \,\overline{\cup}\, tnsh_2$$

■

Definition 4.8 (Negative project, $\overline{\pi}$). Given a negative set $tnsh$ and the desired bit positions to project Υ, *Negative Project* is defined in [14] as

$$\overline{\pi}_{\Upsilon}(tnsh) = \{x | (x \mathcal{M} w) \wedge (\forall w \in \mathcal{U}_{\Upsilon}, \forall z \in \mathcal{U}_{\overline{\Upsilon}}, \ \exists y \in tnsh(y[\Upsilon] \mathcal{M} w \wedge y[\overline{\Upsilon}] \mathcal{M} z))\}$$

e.g., the resulting negative set will contains strings that has a bit value projected in column(s) specified by Υ if and only if all possible binary combination of all strings created with the projected column(s) appear in the negative set. For example, given $tnsh = \{000, 011, 10*, 11*\}$, the $\overline{\pi}_{\Upsilon=1,2}(tnsh) = \{10, 11\}$.

■

Definition 4.9 (Negative projection, $\overline{tnsh|_t}$). The *negative projection* is a function $\overline{tnsh|_t}$: $tNSH^l \times Term \rightarrow tNSH^k$ ($k \leq l$) that selects elements of $tnsh$ projected onto the binary representation of $t \in Term$ and is defined as

$$\overline{tnsh|_t} = \overline{\pi}(tnsh, \Upsilon_t)$$

where Υ_t is equal to all i^{th}-bit positions of $\hat{t}$ and $\hat{t}[i] = 1$. ∎

Example 4.3 (Negative projection). Let $tnsh = \{11^{**}, 1^{*}1^{*}, ^{*}11^{*}, ^{**}11\}$ be the same sharing set as in Example 4.1. The negative projection of $tnsh$ over the term $t = f(X_1, X_2, X_3)$ is $\overline{tnsh|_t} = \{11^{*}, 1^{*}1, ^{*}11\}$. String $^{**}1$ is not in the result because it represents the following strings when fully specified $\{001, 011, 101, 111\}$ and not all these strings are in the complement, e.g., 001 is in the positive result of the same projection over bsh.

5 Experimental Results

We developed a proof-of-concept implementation, which is currently being optimized, in order to measure experimentally the relative efficiency in terms of running time and memory usage obtained with the two new representations described earlier, tSH and $tNSH$. The prototype uses *Patricia tries* [25] to handle efficiently binary and ternary strings, and is connected to a naive *bottom-up* fixpoint analyzer.

Our first objective is to study the implications of the conversions in the representation for analysis. Note that although both tSH and $tNSH$ do not imply a loss of precision, the sizes of the resulting representations and their conversion times can vary significantly from one to another. An essential issue is to determine experimentally the best overall k parameter for the conversion algorithms. Second, we study the core abstract operation of the traditional set-sharing, $amgu$, under two different metrics. One is the running time to perform the abstract unification. The other metric expresses the memory usage through the size of the representation in terms of number of strings during key steps in the unification. All experiments have been conducted on an Intel^R Core^{TM} Duo CPU T2350 at 1.86GHz with 1GB of RAM running Ubuntu 7.04, and were performed with 12-bit strings since we consider this value large enough to show all the relevant features of our approach. In general, within some upper bound, the more variables considered the better the expected efficiency.

The first experiment determines the best k value suitable for the conversion algorithms, shown in Figs. 2 and 3. We proceed by submitting a set of 12-bit strings in random order using different k values. We evaluate size for the smallest output

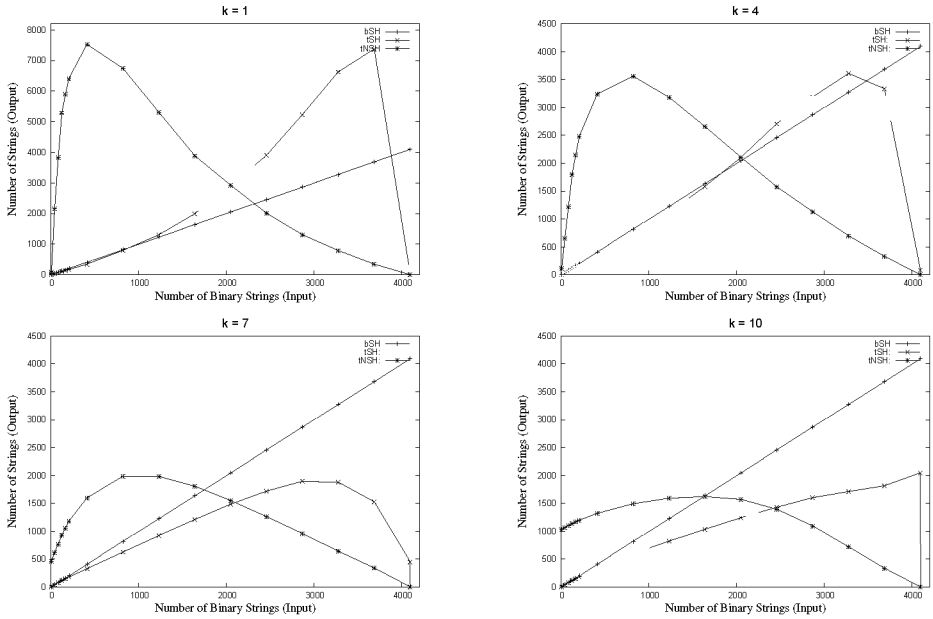

Figure 4: Level of compression after conversions from bSH to tSH and $tNSH$ for k = 1, 4, 7, and 10.

(see Fig. 4) for a given k value. As expected, bSH ($x = y$ line) results in no compression; tSH slowly increases with increasing input size, remaining below bSH (for $k = 7$ and $k = 10$) due to the compression provided by the $*$ symbol and by having little redundancy; $tNSH$, the complement set, starts larger than bSH but quickly tapers off as the input size increases past 50% of $|\mathcal{U}|$. Since the k parameter helps determine the minimum number of specified bits in the set, there is a direct relationship between the k parameter and the size of the output due to compression by the $*$ symbol. A smaller k value, i.e., $k = 1$, introduces the maximum number of $*$ symbols in the set. However, for a given input, a small k value does not necessarily result in the best compression factor (see $k = 1$ of Fig. 4). This result may be counter-intuitive, but it is due to the potentially larger number of unmatched strings that must be re-inserted back into the set determined by all the strings that must be represented by the converted result, see line 13-17 of Fig. 3. In addition, a small k value may result in a set with more ternary strings than the number of binary strings represented. This occurs when multiple ternary strings, none of which subsumes any other, represent the same binary string. This redundancy in the ternary representation is not prevented by ManagedGrowth, and is apparent in Fig. 4 when $|tSH|$ and $|tNSH|$ exceed the maximum size of binary sharing relationships (i.e., 4096). One way to reduce the number of redundant strings is to sort

the binary input by *Hamming distance* before conversion. In the subsequent tests, sorting was performed to maximize compression. We have found empirically that a k setting near (or slightly larger than) $l/2$ is the best overall value considering both the result size and time complexity. We use $k = 7$ in the following experiments. It is interesting to note that a k value of $log_2(l)$ results in polynomial time conversion of the input (see the Complexity column of Table 1) but it may not result in the maximum compression of the set (see $k = 4$ of Fig. 4). Therefore, k may be adjusted to produce results based on acceptable performance level depending on which parameter is more important to the user, the level of compression (memory constraints) or execution time.

Our second experiment shows the comparison in terms of memory usage (Fig. 5, left) and running time (Fig. 5, right) of the conversion algorithms for transforming an initial set of binary strings, bSH, into its corresponding set of ternary strings, tSH, or its complement (negative), $tNSH$. We generated random sets of binary strings (over 30 runs) using $k = 7$ and we converted the set of binary strings using the Convert algorithm described in Fig. 2 for tSH, and NegConvertMissing in Fig. 3 for $tNSH$. We also reduced the number of redundant strings by sorting them using the Hamming distance before conversion. The plot on the left shows that the number of positive ternary strings,$|tSH|$, used for encoding the input binary strings always remains below $|bSH|$, and this number increases slowly with increasing input size. It important to notice that for large values of $|bSH|$, tSH compacts worse than expected and the compression factor is lower. The main cause is the use of the parameter $k = 7$ that implies only the use of 5 or less $*$ symbols for compression. Conversely, the number of negative sharing relationships, $|tNSH|$, is greater than $|bSH|$ and $|tSH|$ up to between 40% and 50%, respectively. However, when the load exceeds those thresholds $tNSH$ compresses much better than its alternatives. For instance, for the maximum number of binary sharing relationships, $tNSH$ compresses them to only one negative string. On the other hand, the rightmost plot shows the average time consumed over 30 runs for both conversion algorithms. Again, $tNSH$ scales better than the positive ternary solution, tSH, after a threshold established around 50% of the maximum number of binary sharing relationships. Our proof-of-concept implementation is not really optimized, since our objective is to study the *relative* performance between the three representations, and thus times are normalized to the range $[0, 1]$. We argue that comparisons that we report between representations are fair since the three cases have been implemented with similar efficiency, and useful since the absolute performance of the base representation is well understood.

Finally, our third experiment shows also the efficiency in terms of the memory usage (in Fig. 6, left) and running time (in Fig. 6, right) when performing the abstract unification for $k = 7$. Several characteristics of the abstract unification

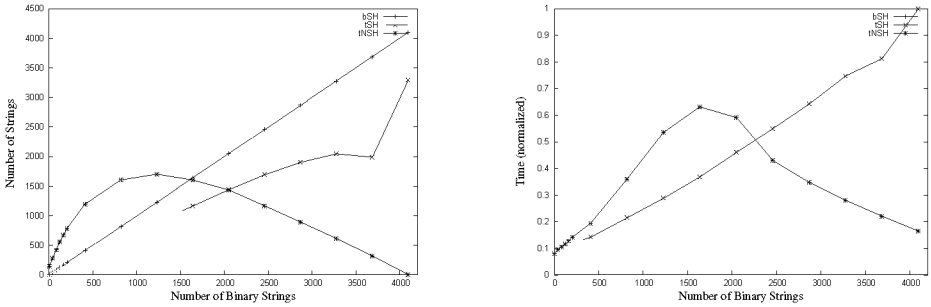

Figure 5: Memory usage (avg. # of strings) and time normalized for conversions with $k = 7$.

influence the memory usage and its performance. Given an arbitrary set of variables of interest $\mathcal{V}$ ($|\mathcal{V}| = 12$), we constructed $x \in \mathcal{V}$ by selecting one variable and $t \in Term$ as a term consisting of a subset of the remaining variables, i.e., $\mathcal{V} \setminus \{x\}$. We tested with different values of t. Another important aspect is the input sharing set, bSH. Again, we reduced the influence of this factor by generating randomly 30 different sets. In the leftmost plot, the x-axis illustrates the number of input binary strings considered during the $amgu$. In the case of the positive and negative ternary $amgu$, the input binary strings were first converted to their corresponding compressed representations. The y-axis shows the number of strings after the unification. The plot shows that exceeding a threshold lower than 500 in the number of input binary sharing relationships, both tSH and $tNSH$ yield a significant smaller number of strings than the binary solution after unification. Moreover, when the number of the input binary strings is smaller than 50% of its maximum value, tSH compresses more efficiently than $tNSH$. However, if this value is exceeded then this trend is reversed: the negative encoding yields a better compression as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. The rightmost plot shows the size of the random binary input sets in the x-axis, and the average time consumed for performing the abstract unification in its y-axis, normalized again from 0 to 1. This graph shows that the execution times behave similarly to the memory usage during abstract unification. Both tSH and $tNSH$ run much faster than bSH. The differences are significant (a factor of 10) for most x-values, reaching a factor of 1000 for large values of $|bSH|$. When the load exceeds a $50 - 60\%$-threshold, $tNSH$ scales better than tSH by a factor of 10. The main difference with respect to the memory usage depicted in the leftmost plot is that for a smaller load, tSH runs as fast as $tNSH$ during unification. The main reason is that the ternary relevant and irrelevant sharing operations are less efficient than their negative counterparts: intersection is an expensive operation in the positive ternary representation whereas

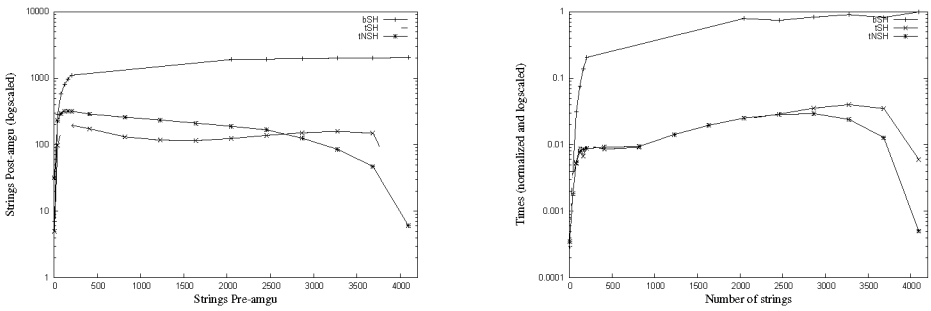

Figure 6: Memory usage (avg. # of strings) and time normalized for amgu over 30 runs with $k = 7$.

the negative intersection is very efficient (positive union).

6 Conclusions and future work

We have presented a novel approach to Set-Sharing that leverages the complement or negative sharing relationships of the original sharing set, without any loss of accuracy. In this work, we based the negative representation on ternary strings. We also showed that the same ternary representation can be used as a positive encoding to efficiently compact the original binary sharing set. This provides the user or the analyzer the option of working with whichever set sharing representation is more efficient for a given problem instance. The capabilities of our negative approach to compress sharing relationships are orthogonal to the use of the ternary representation. Hence, the negative relationships may be encoded by any other representation such as, e.g., Binary Decision Diagrams. Concretely, *Zero-suppressed Binary Decision Diagrams* (ZBDDs) [18] are particularly interesting because ZBDDs were designed to represent sets of combinations (i.e., sets of sets). In addition, this approach may be also applicable to similar sharing-related analyses in object-oriented languages (e.g., [24]).

Our experimental evaluation has shown that our approach can reduce significantly the memory usage of the sharing relationships and the running time of the abstract operations, including the abstract unification. Our experiments also show how to set up key parameters in our algorithms in order to control the desired compression and time complexities. We have shown that we can obtain a reasonable compression in polynomial time by tuning appropriately those parameters. Thus, we believe our results can contribute to the practical, scalable application of Set-Sharing.

We are currently developing a more sophisticated implementation of our do-

mains. We are also integrating our domains into a more sophisticated framework analysis (e.g., `CiaoPP` [16]) in order to take advantage of its efficiency techniques.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.

[2] R. Bagnara, R. Gori, P. M. Hill, and E. Zaffanella. Finite-tree analysis for constraint logic-based languages. *Information and Computation*, 193(2):84–116, 2004.

[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 277(1-2):3–46, 2002.

[4] M. Bruynooghe, M. Codish, and A. Mulkers. Abstract unification for a composite domain deriving sharing and freeness properties of program variables. In F.S. de Boer and M. Gabbrielli, editors, *Verification and Analysis of Logic Languages*, pages 213–230, 1994.

[5] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[6] F. Bueno and M. García de la Banda. Set-Sharing is not always redundant for Pair-Sharing. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, Heidelberg, Germany, April 2004. Springer-Verlag.

[7] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *1994 International Symposium on Logic Programming*, 1993.

[8] M. Codish, V. Lagoon, and F. Bueno. An algebraic approach to sharing analysis of logic programs. In *Proc. of the Fourth International Static Analysis Symposium*, number 1302 in LNCS, pages 68–82. Springer Verlag, 1997.

[9] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[10] Michael Codish, Dennis Dams, Gilberto Filé, and Maurice Bruynooghe. On the design of a correct freeness analysis for logic programs. *The Journal of Logic Programming*, 28(3):181–206, 1996.

[11] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[12] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[13] F. Esponda, E. S. Ackley, S. Forrest, and P. Helman. On-line negative databases (with experimental results). *International Journal of Unconventional Computing*, 1(3):201–220, 2005.

[14] F. Esponda, E. D. Trias, E. S. Ackley, and S. Forrest. A relational algebra for negative databases. Technical Report TR-CS-2007-18, University of New Mexico, 2007.

[15] Christian Fecht. An efficient and precise sharing domain for logic programs. In Herbert Kuchen and S. Doaitse Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 469–470. Springer, 1996.

[16] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[17] P. M. Hill, E. Zaffanella, and R. Bagnara. A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. *Theory and Practice of Logic Programming*, 4(3):289–323, 2004.

[18] Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993.

[19] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[20] A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming*. MIT Press, June 1994.

[21] A. Langen. *Advanced techniques for approximating variable aliasing in Logic Programs*. PhD thesis, Computer Science Dept., University of Southern California, 1990.

[22] X. Li, A. King, and L. Lu. Collapsing Closures. In *ICLP'06*, pages 148–162. Springer-Verlag.

[23] X. Li, A. King, and L. Lu. Lazy Set-Sharing Analysis. In *International Symposium on Functional and Logic Programming, 2006*. Springer-Verlag.

[24] M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[25] Donald R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[26] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

[27] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *ICLP*, 1991.

[28] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[29] Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs*. PhD thesis, University of Texas at Austin, August 1991.

[30] J. Navas, F. Bueno, and M. Hermenegildo. Efficient top-down set-sharing analysis using cliques. In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[31] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[32] E. Zaffanella, R. Bagnara, and P. M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–432. Springer-Verlag, Berlin, 1999.